"""

echo\_chamber\_detector.py

Detecting Echo Chambers and Information Silos in Social Networks.

Input: CSV file with tweets/interactions. Required columns (examples):

- tweet\_id

- user\_id

- text

- in\_reply\_to\_user\_id (optional)

- retweeted\_user\_id (optional)

- timestamp (optional)

Output:

- Community assignments for users

- Community-level statistics (modularity contributions, sentiment homogeneity, topic similarity)

- Echo Chamber Index (ECI) per community

"""

import os

import math

from collections import defaultdict

from itertools import combinations

from typing import Dict, List, Tuple

import numpy as np

import pandas as pd

import networkx as nx

import community as community\_louvain # python-louvain

from sklearn.metrics.pairwise import cosine\_similarity

from sentence\_transformers import SentenceTransformer

from bertopic import BERTopic

from transformers import pipeline

import spacy

from tqdm import tqdm

# -------------------------

# Config

# -------------------------

DATA\_CSV = "tweets.csv" # input CSV path

USER\_MIN\_CONNECTIONS = 3 # filter threshold

EDGE\_WEIGHT\_TYPE = "frequency" # how to aggregate multiple interactions

EMBEDDING\_MODEL = "all-MiniLM-L6-v2" # sentence-transformers model

SENTIMENT\_MODEL = "cardiffnlp/twitter-roberta-base-sentiment" # or use simpler

BERTopic\_REP\_MODEL = EMBEDDING\_MODEL # embedding model used by BERTopic

ALPHA = 0.5 # weighting for modularity in ECI

BETA = 0.5 # weighting for semantic part in ECI

# -------------------------

# Utility & preprocessing

# -------------------------

nlp = spacy.load("en\_core\_web\_sm", disable=["ner", "parser"])

def clean\_text(text: str) -> str:

if pd.isna(text):

return ""

txt = str(text)

# basic cleaning: remove URLs, extra whitespace

txt = txt.replace("\n", " ")

# remove URLs

txt = " ".join([w for w in txt.split() if not w.startswith("http")])

# lowercase

txt = txt.strip()

return txt

def preprocess\_texts(texts: List[str]) -> List[str]:

out = []

for doc in nlp.pipe(texts, batch\_size=64, n\_process=1):

tokens = [token.lemma\_.lower() for token in doc

if not token.is\_stop and token.is\_alpha and len(token) > 1]

out.append(" ".join(tokens))

return out

# -------------------------

# Graph construction

# -------------------------

def build\_user\_graph(df: pd.DataFrame) -> nx.DiGraph:

"""

Build a directed weighted user-user interaction graph from tweets DataFrame.

We consider retweets and replies as edges from source -> target.

"""

G = nx.DiGraph()

# Add nodes for all users

users = pd.concat([df['user\_id'],

df['in\_reply\_to\_user\_id'].fillna(np.nan),

df['retweeted\_user\_id'].fillna(np.nan)]).dropna().unique()

for u in users:

G.add\_node(u)

# Add edges

# Retweets

if 'retweeted\_user\_id' in df.columns:

rt = df.dropna(subset=['retweeted\_user\_id'])[['user\_id','retweeted\_user\_id']]

for \_, row in rt.iterrows():

a, b = row['user\_id'], row['retweeted\_user\_id']

if a == b:

continue

if G.has\_edge(a,b):

G[a][b]['weight'] += 1

else:

G.add\_edge(a,b, weight=1)

# Replies

if 'in\_reply\_to\_user\_id' in df.columns:

rp = df.dropna(subset=['in\_reply\_to\_user\_id'])[['user\_id','in\_reply\_to\_user\_id']]

for \_, row in rp.iterrows():

a, b = row['user\_id'], row['in\_reply\_to\_user\_id']

if a == b:

continue

if G.has\_edge(a,b):

G[a][b]['weight'] += 1

else:

G.add\_edge(a,b, weight=1)

return G

def filter\_low\_degree\_nodes(G: nx.DiGraph, min\_deg: int) -> nx.DiGraph:

G2 = G.copy()

removed = [n for n, d in G2.degree() if d < min\_deg]

G2.remove\_nodes\_from(removed)

return G2

# -------------------------

# Community detection & modularity contribution

# -------------------------

def detect\_communities\_louvain(G: nx.Graph) -> Dict:

"""

Return partition dict: node -> community\_id

"""

# Louvain expects undirected graphs for community detection commonly, but handles weights.

if isinstance(G, nx.DiGraph):

G\_und = G.to\_undirected(reciprocal=False)

else:

G\_und = G

partition = community\_louvain.best\_partition(G\_und, weight='weight')

return partition

def community\_modularity\_contribution(G: nx.Graph, partition: Dict) -> Dict[int, float]:

"""

Compute modularity contribution Q\_k for each community:

Q\_k = (l\_k / m) - (d\_k / (2m))^2

l\_k = sum of internal edge weights within community

d\_k = sum of degrees (weights) of nodes in community (sum of node strengths)

m = total edge weight in graph

"""

# ensure undirected weights for modularity formula

if isinstance(G, nx.DiGraph):

G\_und = G.to\_undirected(reciprocal=False)

else:

G\_und = G

# total edge weight (m) is sum(w)/2 for undirected if edges stored twice, but here to\_undirected merges

total\_weight = sum(data.get('weight',1) for u, v, data in G\_und.edges(data=True))

if total\_weight == 0:

total\_weight = 1.0

# group nodes by community

communities = defaultdict(list)

for node, cid in partition.items():

communities[cid].append(node)

contributions = {}

for cid, nodes in communities.items():

# l\_k: sum of internal edges weights

l\_k = 0.0

d\_k = 0.0

for u in nodes:

d\_k += G\_und.degree(u, weight='weight')

for v in G\_und[u]:

if v in nodes:

l\_k += G\_und[u][v].get('weight', 1)

# Since we iterated edges from u, internal edges counted twice for undirected graph; divide by 2

l\_k = l\_k / 2.0

# modularity contribution formula

m = total\_weight

Q\_k = (l\_k / m) - ( (d\_k / (2\*m))\*\*2 )

contributions[cid] = float(Q\_k)

return contributions

# -------------------------

# NLP: sentiment and embeddings

# -------------------------

def compute\_sentiment\_scores(texts: List[str], sentiment\_model\_name: str = None) -> List[float]:

"""

Return sentiment score per text in [-1, 1] (negative -> -1, positive -> +1)

Uses HuggingFace pipeline for sentiment (three-label). If model not available, fallback to neutral 0.

"""

if sentiment\_model\_name is None:

# fallback: neutral predictions

return [0.0 for \_ in texts]

# Use transformers pipeline

try:

sentiment\_pipe = pipeline("sentiment-analysis", model=sentiment\_model\_name, device=0 if os.getenv("CUDA\_VISIBLE\_DEVICES") else -1)

except Exception:

# device fallback and smaller model options could be used.

sentiment\_pipe = pipeline("sentiment-analysis", model=sentiment\_model\_name)

scores = []

for text in texts:

if not text:

scores.append(0.0)

continue

res = sentiment\_pipe(text[:512])[0] # truncate

label = res.get('label', '').lower()

# Many models return 'LABEL\_0' etc; handle common models

score = res.get('score', 0.0)

if 'negative' in label or 'neg' in label or label.startswith('label\_0'):

scores.append(-1.0 \* score)

elif 'positive' in label or 'pos' in label or label.startswith('label\_2'):

scores.append(1.0 \* score)

else:

# neutral

scores.append(0.0)

return scores

def compute\_user\_embeddings(df: pd.DataFrame, user\_col='user\_id', text\_col='text', model\_name=EMBEDDING\_MODEL) -> Dict:

"""

Compute per-user embedding as the average embedding of their texts.

Returns dict: user -> embedding (numpy array)

"""

model = SentenceTransformer(model\_name)

user\_texts = defaultdict(list)

for \_, row in df[[user\_col, text\_col]].iterrows():

user\_texts[row[user\_col]].append(str(row[text\_col]))

user\_embeddings = {}

for user, texts in user\_texts.items():

# optionally sample if too many texts per user

embeddings = model.encode(texts, show\_progress\_bar=False)

emb\_mean = np.mean(embeddings, axis=0)

user\_embeddings[user] = emb\_mean

return user\_embeddings

# -------------------------

# Topic modeling (BERTopic)

# -------------------------

def fit\_bertopic\_model(df: pd.DataFrame, text\_col='text', embedding\_model=BERTopic\_REP\_MODEL) -> Tuple[BERTopic, List[int], np.ndarray]:

"""

Fit BERTopic on the document corpus and return:

- topic\_model instance

- list of topic ids for each doc

- embeddings (numpy array) for each document

"""

embedder = SentenceTransformer(embedding\_model)

docs = df[text\_col].astype(str).tolist()

embeddings = embedder.encode(docs, show\_progress\_bar=True)

topic\_model = BERTopic(embedding\_model=None) # we will pass embeddings directly

topics, \_ = topic\_model.fit\_transform(docs, embeddings)

return topic\_model, topics, np.array(embeddings)

# -------------------------

# Community semantic stats: sentiment homogeneity & topic similarity

# -------------------------

def compute\_sentiment\_homogeneity\_by\_community(df: pd.DataFrame,

partition: Dict,

sentiment\_scores: List[float]) -> Dict[int, float]:

"""

Given sentiment\_scores aligned with df rows, compute homogeneity per community.

SH(C\_k) = 1 - (var(S\_k) / max\_var) where max\_var is global variance across communities (or max observed)

Returns dict: community\_id -> SH in [0,1]

"""

# map each row to a community via user

df = df.reset\_index(drop=True)

community\_sents = defaultdict(list)

for i, row in df.iterrows():

user = row['user\_id']

if user not in partition:

continue

community\_sents[partition[user]].append(sentiment\_scores[i])

# compute variances

variances = {}

for cid, sents in community\_sents.items():

if len(sents) <= 1:

variances[cid] = 0.0

else:

variances[cid] = float(np.var(sents))

# normalization: use max variance across communities or 1.0

max\_var = max(variances.values()) if variances else 0.0

if max\_var <= 0:

# all zero variance => perfect homogeneity

sh = {cid: 1.0 for cid in variances.keys()}

return sh

sh = {cid: 1.0 - (v / max\_var) for cid, v in variances.items()}

# clamp

sh = {cid: max(0.0, min(1.0, val)) for cid, val in sh.items()}

return sh

def compute\_topic\_similarity\_by\_community(user\_embeddings: Dict, partition: Dict) -> Dict[int, float]:

"""

For each community, compute average pairwise cosine similarity of user embeddings.

Returns dict: community\_id -> TS in [0,1].

If a community has a single user, TS = 1.0 (maximal apparent similarity).

"""

community\_embs = defaultdict(list)

for user, emb in user\_embeddings.items():

if user in partition:

community\_embs[partition[user]].append(emb)

ts = {}

for cid, embs in community\_embs.items():

if len(embs) == 0:

ts[cid] = 0.0

continue

if len(embs) == 1:

ts[cid] = 1.0

continue

arr = np.vstack(embs)

sim\_mat = cosine\_similarity(arr)

# average of upper triangle excluding diagonal

n = arr.shape[0]

idxs = np.triu\_indices(n, k=1)

if len(idxs[0]) == 0:

ts[cid] = 1.0

else:

avg\_sim = float(np.mean(sim\_mat[idxs]))

# map to [0,1] if negative possible; cosine similarity here should be [-1,1], but embeddings small models usually give [0,1]

ts[cid] = float((avg\_sim + 1.0) / 2.0) if avg\_sim < 0 else float(avg\_sim)

# clamp

ts[cid] = max(0.0, min(1.0, ts[cid]))

return ts

# -------------------------

# ECI calculation

# -------------------------

def compute\_ECI\_for\_communities(Qk: Dict[int, float], SH: Dict[int, float], TS: Dict[int, float], alpha=ALPHA, beta=BETA) -> Dict[int, float]:

"""

ECI(C\_k) = alpha \* Q\_k + beta \* (SH + TS)

Q\_k may be negative for some formulations; we will normalize Q\_k to [0,1] across communities before combining.

"""

# Normalize Qk across communities to [0,1]

all\_q = np.array(list(Qk.values())) if Qk else np.array([0.0])

q\_min, q\_max = float(all\_q.min()), float(all\_q.max())

q\_range = q\_max - q\_min if q\_max > q\_min else 1.0

Qk\_norm = {cid: (q - q\_min) / q\_range for cid, q in Qk.items()}

eci = {}

for cid in set(list(Qk.keys()) + list(SH.keys()) + list(TS.keys())):

qn = Qk\_norm.get(cid, 0.0)

sh = SH.get(cid, 0.0)

ts = TS.get(cid, 0.0)

eci\_val = alpha \* qn + beta \* ( (sh + ts) / 2.0 ) # average semantic part

# ensure in [0,1]

eci[cid] = float(max(0.0, min(1.0, eci\_val)))

return eci

# -------------------------

# Main pipeline

# -------------------------

def run\_pipeline(csv\_path=DATA\_CSV, min\_connections=USER\_MIN\_CONNECTIONS):

print("Loading data...")

df = pd.read\_csv(csv\_path)

# Basic cleaning

df['text'] = df['text'].fillna("").astype(str).apply(clean\_text)

print(f"Total rows: {len(df)}, unique users: {df['user\_id'].nunique()}")

# Build graph

print("Building interaction graph...")

G = build\_user\_graph(df)

print(f"Graph nodes: {G.number\_of\_nodes()}, edges: {G.number\_of\_edges()}")

# Filter low-degree nodes

print(f"Filtering nodes with degree < {min\_connections} ...")

Gf = filter\_low\_degree\_nodes(G, min\_connections)

print(f"Filtered graph nodes: {Gf.number\_of\_nodes()}, edges: {Gf.number\_of\_edges()}")

# Community detection

print("Running Louvain community detection...")

partition = detect\_communities\_louvain(Gf)

num\_comms = len(set(partition.values()))

print(f"Detected {num\_comms} communities")

# Compute modularity contributions

print("Computing modularity contributions per community...")

Qk = community\_modularity\_contribution(Gf, partition)

# Sentiment analysis across documents

print("Computing sentiment scores (this may take time)...")

sentiments = compute\_sentiment\_scores(df['text'].tolist(), sentiment\_model\_name=SENTIMENT\_MODEL)

# Note: if sentiment model too slow, use a subset or a simpler lexicon-based method

# Compute user embeddings

print("Computing user embeddings (sentence-transformers)...")

user\_embeddings = compute\_user\_embeddings(df, user\_col='user\_id', text\_col='text', model\_name=EMBEDDING\_MODEL)

# Topic modeling (optional, we use user embeddings for topic-similarity measurement)

# topic\_model, topics, doc\_embs = fit\_bertopic\_model(df, text\_col='text')

# Sentiment homogeneity

print("Computing sentiment homogeneity by community...")

SH = compute\_sentiment\_homogeneity\_by\_community(df, partition, sentiments)

# Topic similarity by community

print("Computing topic similarity by community...")

TS = compute\_topic\_similarity\_by\_community(user\_embeddings, partition)

# Compute ECI

print("Computing Echo Chamber Index (ECI) for each community...")

eci = compute\_ECI\_for\_communities(Qk, SH, TS, alpha=ALPHA, beta=BETA)

# Assemble results table

print("Assembling results...")

comm\_stats = []

for cid in sorted(eci.keys()):

comm\_stats.append({

"community\_id": cid,

"Qk": Qk.get(cid, 0.0),

"SH": SH.get(cid, 0.0),

"TS": TS.get(cid, 0.0),

"ECI": eci.get(cid, 0.0),

"num\_users": sum(1 for u, c in partition.items() if c == cid)

})

results\_df = pd.DataFrame(comm\_stats).sort\_values("ECI", ascending=False)

print(results\_df.head(20).to\_string(index=False))

# Save outputs

results\_df.to\_csv("community\_ECI\_results.csv", index=False)

# Also save partition mapping

pd.DataFrame(list(partition.items()), columns=['user\_id','community']).to\_csv("user\_partition.csv", index=False)

print("Done. Results saved to community\_ECI\_results.csv and user\_partition.csv")

return results\_df, partition

# -------------------------

# CLI

# -------------------------

if \_\_name\_\_ == "\_\_main\_\_":

import argparse

parser = argparse.ArgumentParser(description="Detect echo chambers and compute ECI.")

parser.add\_argument("--csv", type=str, default=DATA\_CSV, help="Input CSV path (tweets dataset).")

parser.add\_argument("--min-connections", type=int, default=USER\_MIN\_CONNECTIONS, help="Minimum degree to keep user in graph.")

args = parser.parse\_args()

run\_pipeline(csv\_path=args.csv, min\_connections=args.min\_connections)